Design of Digital Circuits and Systems

Memory I

Instructor: Justin Hsia

Teaching Assistants:

Colton Harris	Deepti Anoop
Gayathri Vadhyan	Jared Yoder
Lancelot Wathieu	Matthew Hung

Relevant Course Information

- hw1 can still be submitted tonight (1 late day)
  - Solution outline will be posted tomorrow (UW login)
- hw2 released today, due next Wednesday (4/10)

- Lab 1 reports due Friday (4/5), demos 4/6-12
- Lab 2 released today, due next Friday (4/12)

- Quiz 1 is Thursday, 4/4 in last 25 min of lecture
  - FSM design – state diagram, explain design decisions
  - 23sp Quiz 1 available for practice on course website

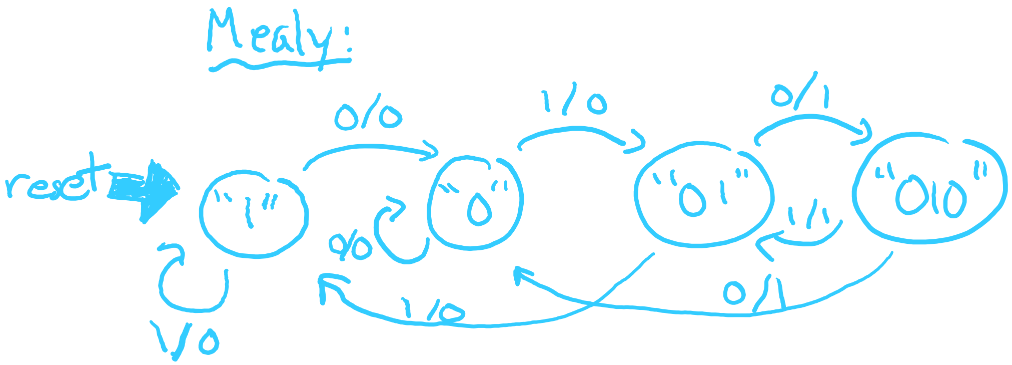
Review Question

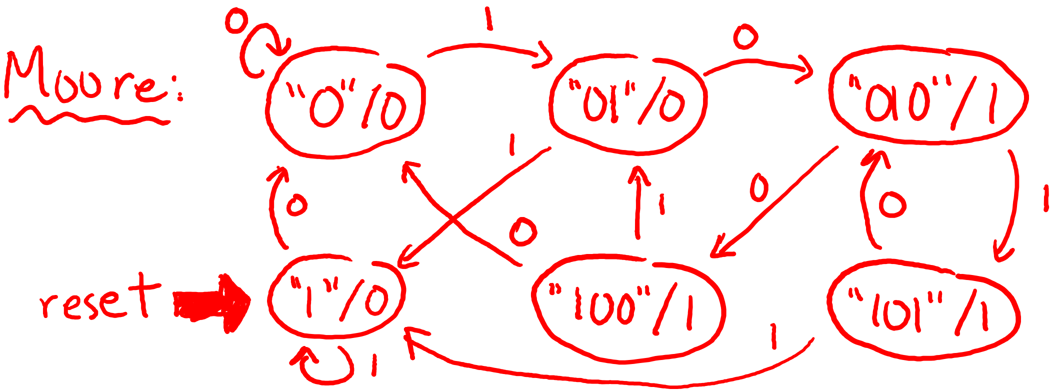
- Design a FSM that will output two consecutive 1's any time it sees the string "010" and outputs 0 otherwise
  - 1 input bit and 1 output bit
  - How many state bits do we need?
  - Which state should be your initial/reset state?

- If you have time, design both a **Moore** machine and **Mealy** machine "from scratch" (*i.e.*, don't convert between the two)
  - Which seems easier to implement? Can you name specific ways that the SystemVerilog implementation will be "easier"?

Review Solution

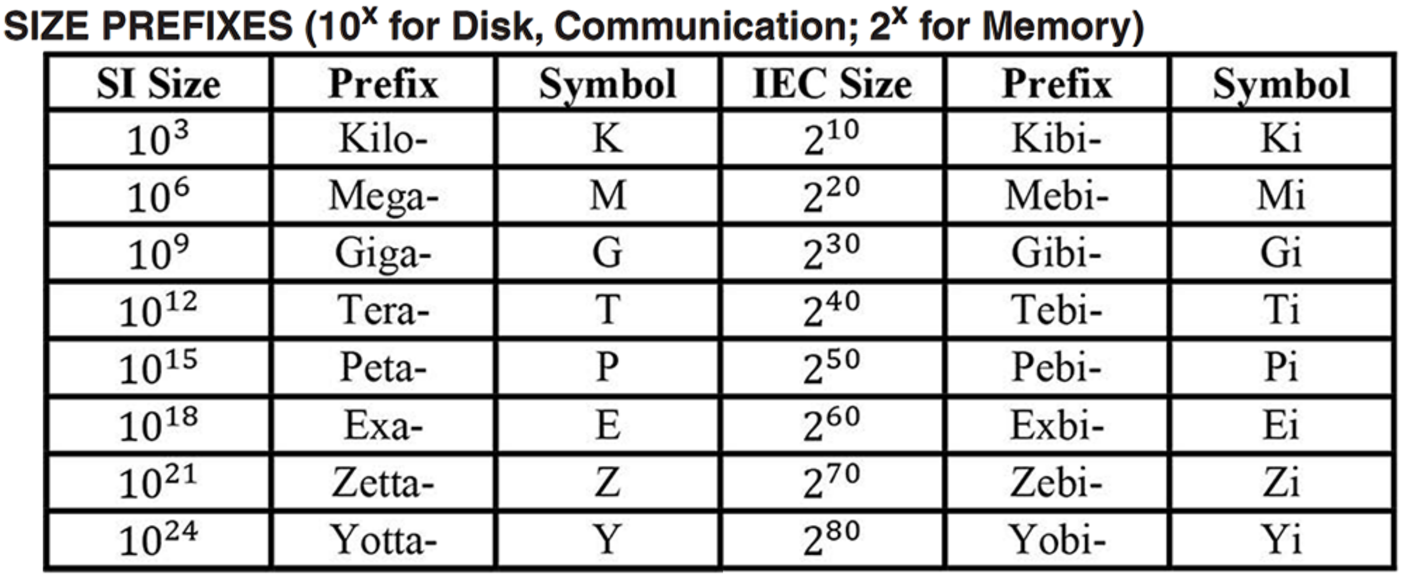
- Design a FSM that will output two consecutive 1's any time it sees the string "010" and outputs 0 otherwise





Aside: Powers of 2 and Prefixes

- Here focusing on large numbers (*i.e.*, exponents > 0)
- SI prefixes are *ambiguous* if base 10 or 2
  - Note that $10^3 \approx 2^{10}$
- IEC prefixes are *unambiguously* base 2



SIZE PREFIXES (10^x for Disk, Communication; 2^x for Memory)

SI Size	Prefix	Symbol	IEC Size	Prefix	Symbol
10^3	Kilo-	K	2^{10}	Kibi-	Ki
10^6	Mega-	M	2^{20}	Mebi-	Mi
10^9	Giga-	G	2^{30}	Gibi-	Gi
10^{12}	Tera-	T	2^{40}	Tebi-	Ti
10^{15}	Peta-	P	2^{50}	Pebi-	Pi
10^{18}	Exa-	E	2^{60}	Exbi-	Ei
10^{21}	Zetta-	Z	2^{70}	Zebi-	Zi
10^{24}	Yotta-	Y	2^{80}	Yobi-	Yi

$2^0 = 1$
$2^1 = 2$
$2^2 = 4$
$2^3 = 8$
$2^4 = 16$
$2^5 = 32$
$2^6 = 64$
$2^7 = 128$
$2^8 = 256$
$2^9 = 512$
$2^{10} = 1024$

Large Powers of 2 and Units

- Because IEC prefixes are powers of 2^{10}, we can convert any large power of 2 as follows:
  - Note that we are only changing the *quantity* and the *units* remain the same

$$2^{XY} \text{“things”} = \left[\begin{array}{l} Y=0 \rightarrow 1 \\ Y=1 \rightarrow 2 \\ Y=2 \rightarrow 4 \\ Y=3 \rightarrow 8 \\ Y=4 \rightarrow 16 \\ Y=5 \rightarrow 32 \\ Y=6 \rightarrow 64 \\ Y=7 \rightarrow 128 \\ Y=8 \rightarrow 256 \\ Y=9 \rightarrow 512 \end{array}\right] + \left[\begin{array}{l} X=0 \rightarrow \\ X=1 \rightarrow \text{Kibi-} \\ X=2 \rightarrow \text{Mebi-} \\ X=3 \rightarrow \text{Gibi-} \\ X=4 \rightarrow \text{Tebi-} \\ X=5 \rightarrow \text{Pebi-} \\ X=6 \rightarrow \text{Exbi-} \\ X=7 \rightarrow \text{Zebi-} \\ X=8 \rightarrow \text{Yobi-} \end{array}\right] + \text{“things”}$$

- Examples:
  - 2 GiB of memory? $2\text{ GiB} = 2^1 \times 2^{30}$ bytes $= 2^{31}$ bytes
  - 2^{47} things into IEC: 2^{47} things $= 2^7 \times 2^{40} =$ 128 Ti-things

Memory

- Several forms of memory are available, which include:
  - ~~Secondary memory (*e.g.*, hard disk, flash drive)~~ not covered in 371
  - Read-only memory (ROM)
  - Random-access memory (RAM)
  - Register files
    - Small, fast, fixed-sized memory that hold CPU data state
  - First in, first out (FIFO) buffers

Embedded FPGA Memory

- An FPGA contains prefabricated memory modules
  - Intended for small or intermediate-sized storage
  - Contents of memory blocks can be configured via memory initialization files (`.mif`)
- The DE1-SoC's Cyclone V FPGA (Cyclone V SE A5) has:
  - 31k Adaptive Logic Modules (ALMs)
  - 4.45 Mbits of memory organized as 397 memory blocks, each with 10 kbits of storage (M10K)
    - Flexible, configurable memory storage available to the designer
    - Each M10K can act as single-port memory, dual-port RAM, shift register, ROM, or a FIFO buffer
  - More info on website: DE1-SoC → Cyclone V Handbook

DE1-SoC Memory

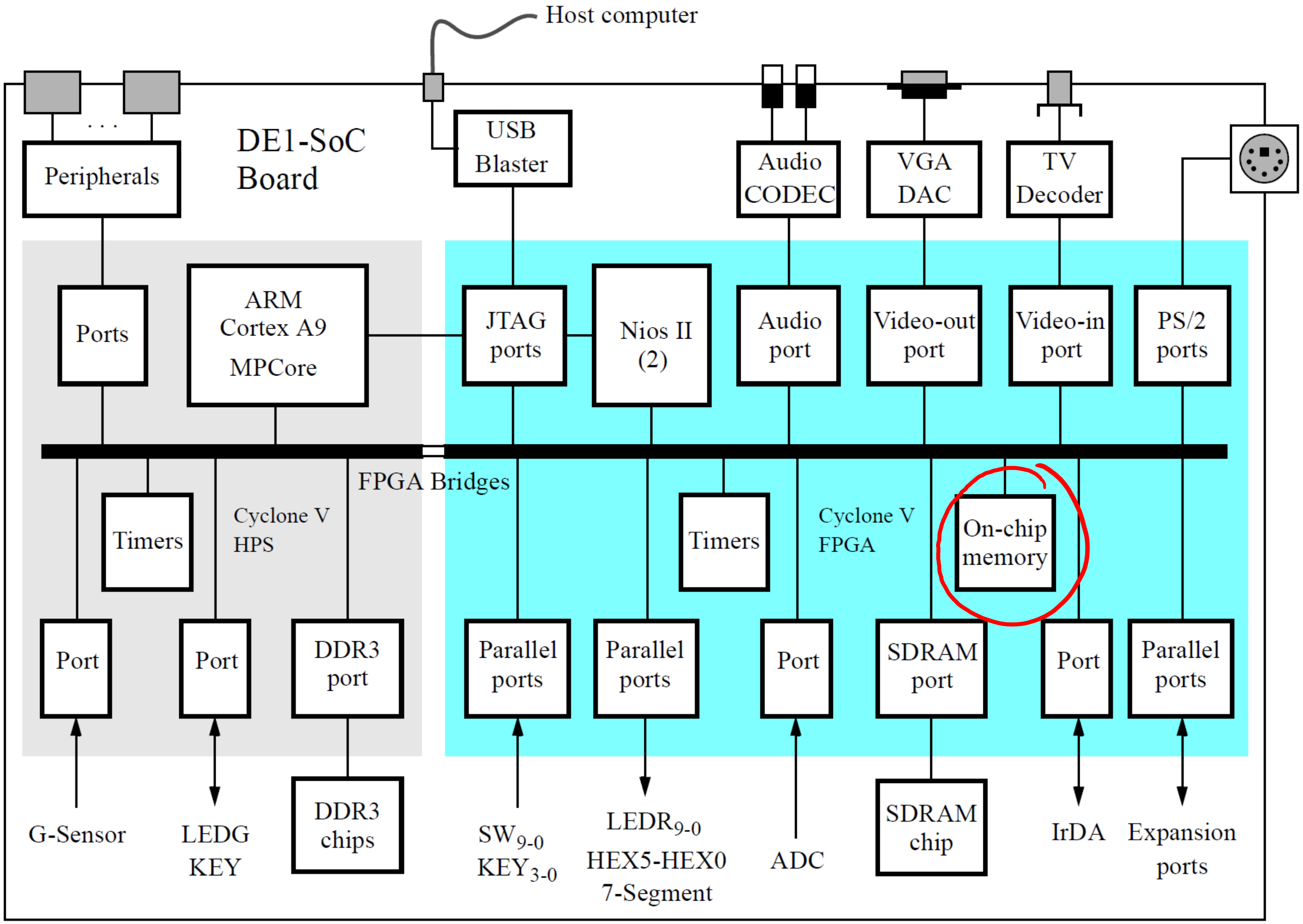
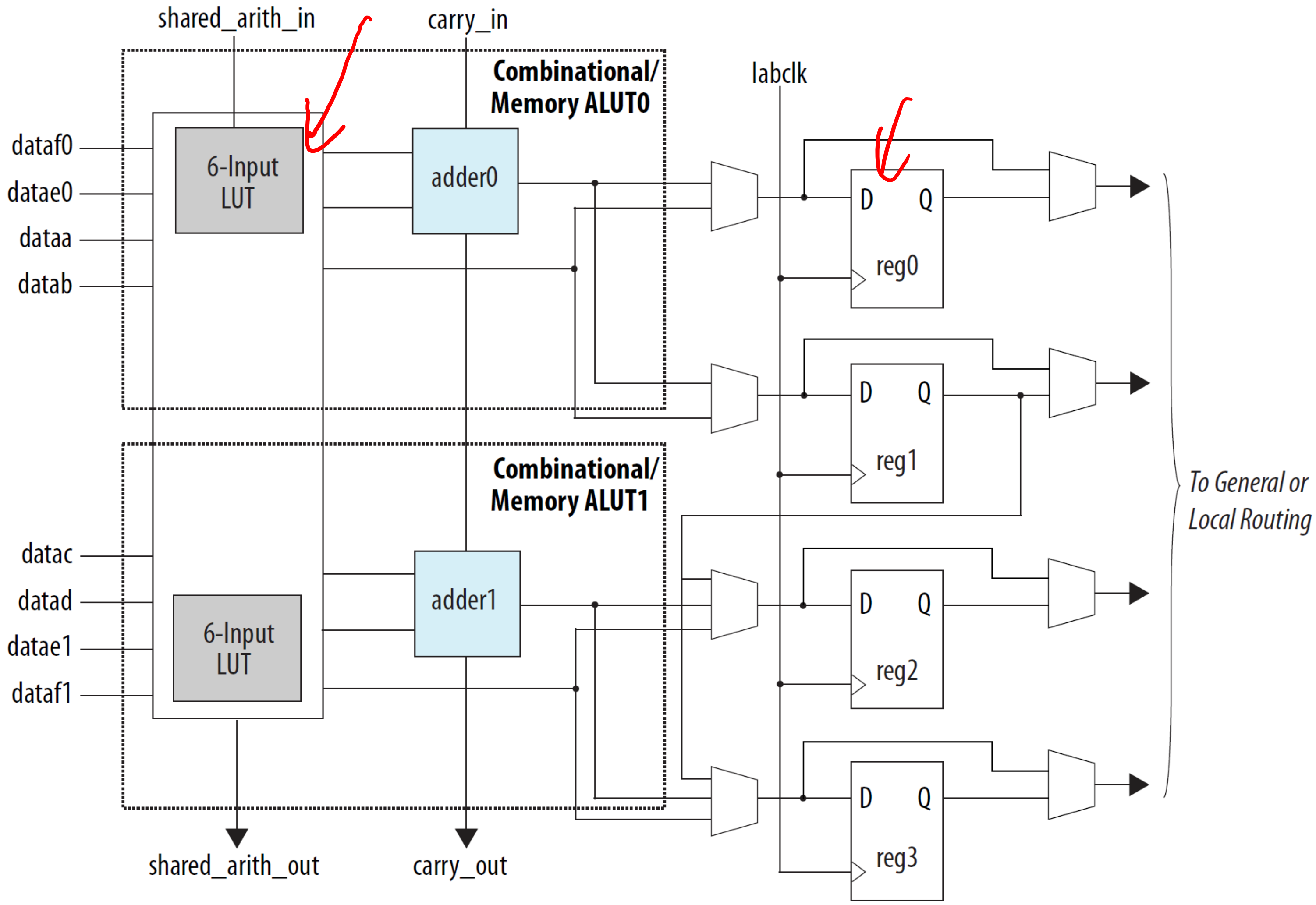


Figure 1. Block diagram of the DE1-SoC Computer.

Cyclone V Adaptive Logic Modules

https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/cyclone-v/cv_5v2.pdf

Figure 1-5: ALM High-Level Block Diagram for Cyclone V Devices

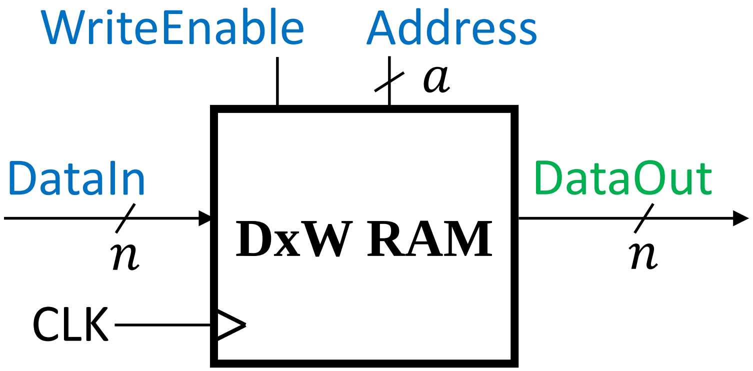


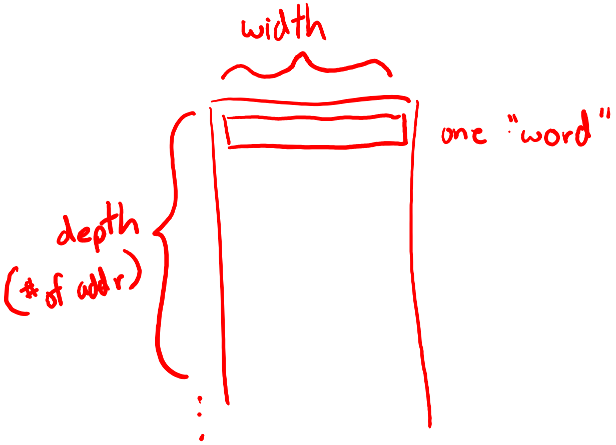
Memory Modules

- Key characteristics of a simple memory module:
  - Width – the number of bits in a word
  - Depth – the number words in the module
  - Number of ports
  - Synchronicity of port access – whether or not accesses are controlled by a clock signal
  - Simultaneous address access – can the same address be used for both read and write operations

size of memory

- Example:





Memory Characteristics

- Memory units are specified as **depth** × **width**.

  # of addresses — size of word

- For the following memory units, answer:

  8 bits = 1 byte

  1) Memory capacity (in bits and bytes using IEC prefixes) depth × width
  2) Width of address bus (a) $\lceil \log_2(\text{depth}) \rceil$
  3) Width of data output bus (n) width

- Memory 1: 8Ki × 32

  2^3 2^{10} 2^5

  ① 2^{18} bits = 256 Kibits = 32 Ki-bytes
  ② 13 bits
  ③ 32 bits

- Memory 2: 2Gi × 8

  2^1 2^{30} 2^3

  ① 2^{34} bits = 16 Gibits = 2 Gibytes
  ② 31 bits
  ③ 8 bits

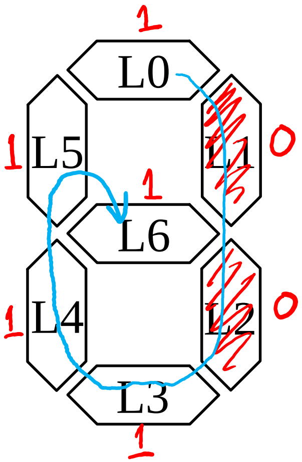
Technology Break

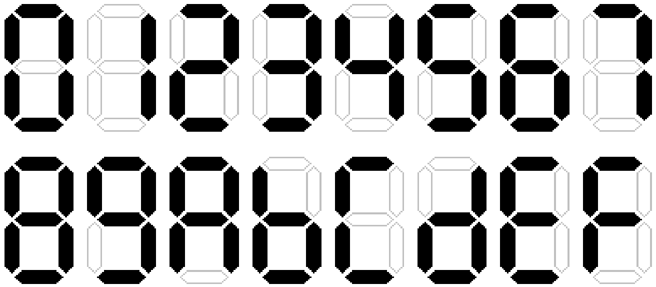
Memory Type #1: ROM

- **R**ead-**O**nly **M**emory
  - A purely *combinational* circuit (no internal state)
  - Output is determined solely by address input
- In an FPGA:
  - No actual embedded ROM, but can be emulated by a combinational circuit or a RAM with the write operation disabled
  - Only practical for small tables
- In SystemVerilog:
  - Define the ROM content as a 2-dimensional constant
  - Oftentimes a selected assignment or case statement

Example ROM

- Hex-to-7seg LED decoder:
  - Segments:
    - Active low





B3	B2	B1	B0	L0	L1	L2	L3	L4	L5	L6
0	0	0	0	0	0	0	0	0	0	1
0	0	0	1	1	0	0	1	1	1	1
0	0	1	0	0	0	1	0	0	1	0
	⋮					⋮				
1	1	0	1	1	0	0	0	0	1	0
1	1	1	0	0	1	1	0	0	0	0
1	1	1	1	0	1	1	1	0	0	0

depth (addresses)

width (words)

- ROM size?

$\boxed{16 \times 7 \text{ bits}}$

Example ROM (SystemVerilog)

```
module ROM_case(addr, data);

   input  logic [3:0] addr;
   output logic [6:0] data;

   always_comb
      case (addr)
         //              L6543210
         4'h0: data = 7'b1000000;
         4'h1: data = 7'b1111001;
         4'h2: data = 7'b0100100;
         // ...
         4'hD: data = 7'b0100001;
         4'hE: data = 7'b0000110;
         4'hF: data = 7'b0001110;
      endcase

endmodule
```

watch the bit ordering!

Reading Data from a File

- Hard-coded values in your SV files can be tedious to format, debug, and swap out!
- Verilog provides *system tasks* to read data from a text file into an array: `$readmemb()` and `$readmemh()`
  - Basic usage: `$readmemb("file.txt", my_array);`
  - Reads in numerals (in **b**inary or **h**ex) from file separated by *whitespace* (*i.e.*, spaces, tabs, newlines)
  - Can avoid some recompilation
  - File can have any extension (.txt, .mem, .bit, .hex)
  - ☆ Be very careful with dimensions of array and ordering of data in file!

Example ROM (SystemVerilog)

```
module ROM_file(addr, data);

   input  logic [3:0] addr;
   output logic [6:0] data;

   logic [6:0] ROM [0:15];  // (!) unpacked dimension

   initial
      // reads binary values from file into array
      $readmemb("seg7decode.txt", ROM);

   assign data = ROM[addr];

endmodule
```

dimensions should match

seg7decode.txt:

read left-to-right, then top-to-bottom.

read as ROM[0]

white space

```
1000000 1111001 0100100 0110000
0011001 0010010 0000010 1111000
0000000 0010000 0001000 0000011
1000110 0100001 0000110 0001110
```

read as ROM[15]

Dynamic Array Indexing Operation

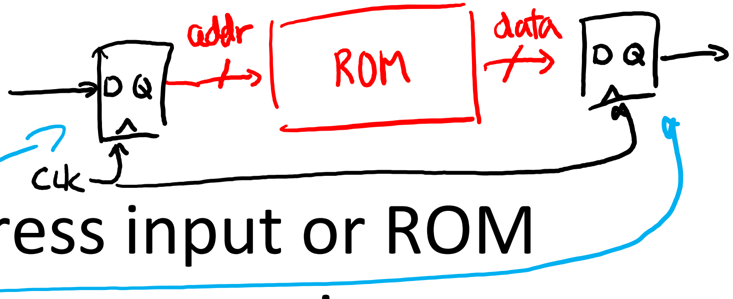
- A signal can be used as an index to access an element in the array:

```
assign data = ROM[addr];
```

equivalent to

```
always_comb
   case (addr)
      4'h0: data = ROM[0];
      4'h1: data = ROM[1];
      // ...
      4'hE: data = ROM[14];
      4'hF: data = ROM[15];
   endcase
```

Synchronicity Revisited



- To synchronize either the address input or ROM output, we can add a register as appropriate
  - Sometimes called "registering" the input or output to make your module "synchronous"
  - In SystemVerilog, can be done inside or outside of the module:

converted assign

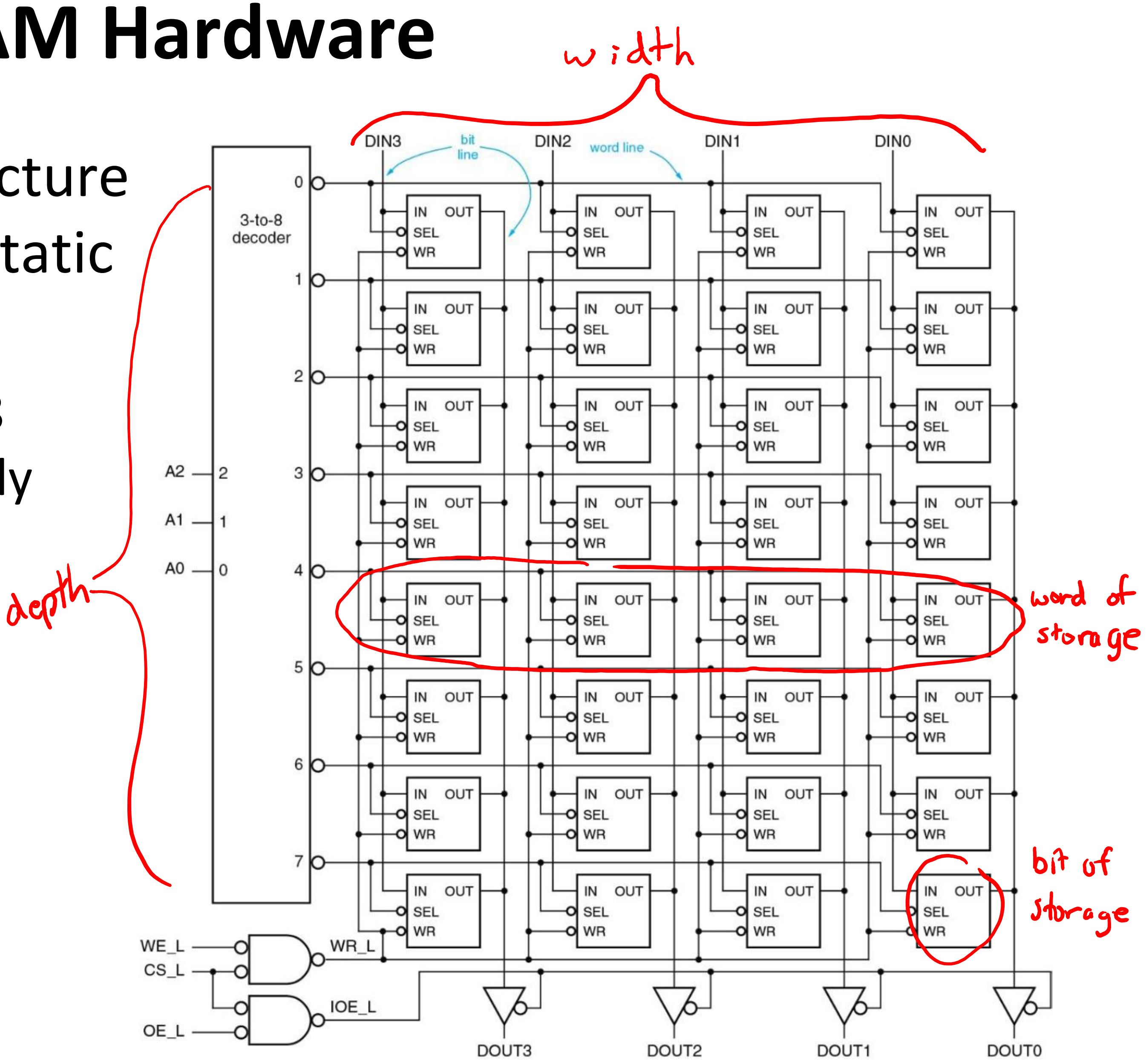
```
    initial
       $readmemb("seg7decode.txt", ROM);

    // internally-registered ROM module
    always_ff @(posedge clk)
       data <= ROM[addr];
```

Memory Type #2: RAM

- **R**andom-**A**ccess **M**emory
  - Can read and write to any address instead of having to read in sequence (*e.g.*, a magnetic disk)
- Can be implemented using different semiconductor technologies:
  - Static RAM (SRAM): uses flip-flops to store data
  - Dynamic RAM (DRAM): uses capacitors and transistors to store data that need to be periodically refreshed
  - SRAM is faster but more expensive than DRAM
- In a typical home computer:
  - CPU registers and caches are SRAM
  - "Memory" is *synchronous* DRAM (SDRAM)

Example RAM Hardware

- Internal structure of an 8 × 4 static RAM:
  - Figure 15.18 from Wakerly



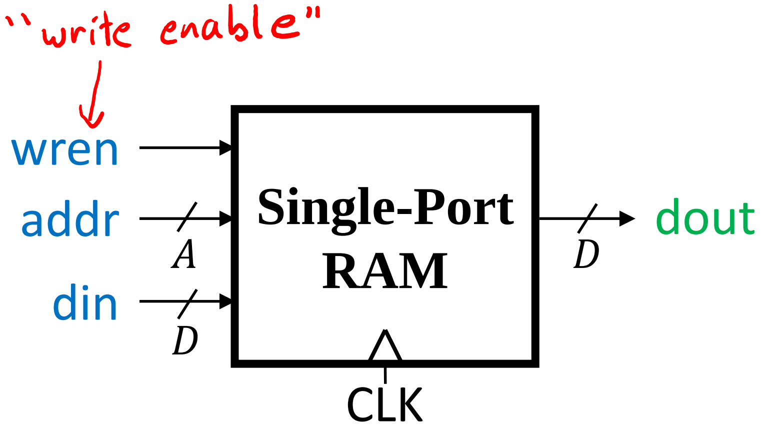
RAM Variants

- Note that not all of the terminology used here is standardized:

1) Synchronicity
   - Are the operations controlled by the clock?
   - Can be applied to reading and writing independently
2) Number of "ports"
   - Not in the same sense as number of SV module ports
   - Can roughly think of as “channels” (set of addr in, data in, write enable, and data out ports) – how many reads and writes can occur simultaneously?
3) Address independence
   - Are the port’s read and write operation based on the same or independent addr in buses?

Synchronous Single-Port RAM

- Synchronous Inputs:
  - wren (1 = write, 0 = read)
  - addr (A-bit address)
  - din (D-bit data)
- Synchronous Output:
  - dout (D-bit data)



A address bits ⟺ 2^A addresses

- Implementation hints:
  - Will need an internal RAM array of what size? $2^A \times D$
  - To synchronize, should update on clock triggers always_ff @(posedge clk)
  - What should dout do when wren = 1? also set to din

Synchronous Single-Port RAM (SV)

```
module RAM_single #(parameter A, D)
                  (clk, wren, addr, din, dout);

   input  logic clk, wren;
   input  logic [A-1:0] addr;
   input  logic [D-1:0] din;
   output logic [D-1:0] dout;
                                    could be either ordering since
                                      we aren't loading from a file
   logic [D-1:0] RAM [0:2**A-1];

   always_ff @(posedge clk) begin
      if (wren) begin        // write
         RAM[addr] <= din;
         dout <= din;
                   can't be RAM[addr] because non-blocking assignment
      end
      else                  // read
         dout <= RAM[addr];
   end  // always_ff

endmodule
```